# 1 Why algorithms?

Q: Can I get a good programming job without knowing something about algorithms and data structures?

A: Yes... but do you really want to be programming GUIs your entire life?

Algorithms are the heart of computer science, and their essential nature is to automate some aspect of collecting, organizing and processing information. Today, information of all kinds is increasingly available in enormous quantities.

However, our ability to make sense of this information, to manage, organize and search it, and to use it for practical purposes, e.g., self-driving cars, adaptive computation, search algorithms for the Internet or for social networks, artificial intelligence, and many scientific applications, relies on the design of *correct* and *efficient* algorithms, that is, algorithms that perform as intended on all inputs and are fast, use little memory and provide guarantees on their performance.

## 1.1 The business pitch

1. Almost all big companies want programmers with knowledge of algorithms: Microsoft, Apple, Google, Facebook, Oracle, IBM, Yahoo, NIST, NOAA, etc.
2. In most programming job interviews, they will ask you several questions about algorithms and/or data structures. They may even ask you to write pseudo or real code on the spot.
3. Your knowledge of algorithms will set you apart from the masses of interviewees who know only how to program.
4. If you want to start your own company, you should know that many startups are successful because they've found better algorithms for solving a problem (e.g. Google, Akamai, etc.).

## 1.2 The intellectual pitch

1. You will improve your research skills in almost any area.
2. You will write better, faster, more elegant code.
3. You will think more clearly, more abstractly and more mathematically.
4. It's one of the most challenging and interesting area of Computer Science.

## 1.3 An example

The following is a question commonly asked in job interviews:

- You are given an array containing integers between 1 and 1,000,000.
- Every integer from 1 and 1,000,000 is in the array once, but one is in the array twice.

Q: Can you determine which integer is in the array twice?

Q: Can you do it while iterating through the array only once?

### 1.3.1 A naïve solution

Here's a simple and intuitive solution for solving this problem:

1. Create a new array $L$ of `int`s between 1 and 1,000,000; we'll use this array to count the occurrences of each number.
2. Initialize all entries to 0.
3. Iterate over the input array; each time a number $i$ is seen, increment the count $L[i]$ in the new array
4. Iterate over $L$ and see which number occurs twice, $L[i] > 1$.
5. Return that number, $i$.

How long does this algorithm take? This algorithm iterates through 1,000,000 numbers 3 times. It also uses twice as much space as the original input sequence. (Maybe $10^6$ doesn't seem too big, but imagine something $10^{10}$ or even bigger.)

If the size of the input array is $n$, we say mathematically that asymptotically, that is, as $n \to \infty$, this algorithm takes $\Theta(n)$ time. We can also specify the amount of additional space the algorithm takes, i.e., the memory beyond what's necessary to store the input array: $O(n)$. That is, both are *linear* in the size of the input sequence, and moreover, they are bounded above and below (a *tight* bound) by linear functions.

In fact, this solution is not efficient: it wastes both time and space. How much better can we do?

The good news is that, while inefficient, this algorithm is *correct*, meaning that on every possible input, it returns the correct answer. To show that this is true, we need to prove that it will never fail on any input. We often use a proof-by-contradiction approach to do this. As a first step to analyzing any algorithm, try to identify a counter example for the claim that the algorithm is correct. If you can prove that no such counter example exists, you are done.

### 1.3.2 A better solution

Recall from discrete mathematics that $\sum_{i=1}^{n} i = n(n+1)/2$. We can use this mathematical fact to make a much more efficient algorithm.

- Let $S$ be the sum of the values in the input array.
- Let $n$ be the largest integer in the array; in this case, $n = 1,000,000$.
- Let $x$ be the value of the repeated number.
- Then, $S = n(n+1)/2 + x$,
- And $x = S - n(n+1)/2$.

Thus, an efficient algorithm is the following:

1. Iterate through the input array, summing the numbers; let $S$ be this sum; let $n$ be the largest value observed in this iteration.
2. Let $x = S - n(n+1)/2$.
3. Return $x$.

How much time does this take? We iterate through the input array exactly once, so it's roughly three times faster than the naïve algorithm. Plus, we use only three constants to store our intermediate work, so this uses much less space. Asymptotically, it takes $O(n)$ time and $O(1)$ space.

## 1.4 The Punch Line

Designing good algorithms matters. As computer scientists, we aim to

1. design *correct* solutions to problems (does not fail on any input),
2. design *elegant* solutions to problems (simplicity is a virtue),
3. design *efficient* solutions (use as few resources as possible), and
4. provide performance guarantees where possible (you have thought carefully about the worst possible behavior).

Achieving these goals is not always easy. Many of the algorithms we'll see in this course are clever, but very few are the most efficient solution possible.

Generally, the first solution that comes to mind will not be either correct or the most efficient. But, if you think carefully about the mathematical structure of the problem, about the inputs that produce bad behavior and apply the tools you learn in this class, you can almost always do better. Your goal for this class should be to sharpen your ability to do these things.

### 1.5 Learning Goals

The goals of this course are several, but they all focus on learning how to *think rigorously* about algorithms, i.e., a computational process that transforms some particular input $X$ into an output $Y$ with specified properties, under a resource budget of $Z$. This will be done in part by learning a variety of specific algorithms and by learning how to analyze them mathematically.

What does "analyze mathematically" mean?

Typically, it means *(i) to mathematically prove that an algorithm $\mathcal{A}$ is correct (i.e., performs as intended on every possible input) and (ii) to mathematically prove that the resources it uses (either space or time) do not exceed some function $f(n)$ for an input of size $n$.*

Our specific goals for this class are as follows. Students will

- become familiar with "standard" algorithms for abstract problem solving;
- learn how to mathematically prove properties of algorithms, including their correctness;
- analyze the time and space complexity of algorithms;
- understand the relative merits or demerits of different algorithms in practice;
- adapt and combine algorithms to solve problems that may arise in practice; and,
- learn common strategies in the design of new algorithms for emerging applications.

## 2 Analyzing algorithms

Designing and analyzing algorithms is not a formulaic domain. It requires careful thinking and some degree of cleverness. The goal of the class is to equip you with a larger and more powerful set of tools that you can use to attack novel problems.

### 2.1 Computing with an abstract "model" of computation

Algorithms are *machine independent*, which means they can be implemented in any Turing-complete language on any computational architecture that is equivalent to a Turing machine. In other words, an algorithm is a procedure that will run on any general purpose computer.

However, there are several different ways to design such a computer. In this class, we will use an abstract model of computation called the *Random Access Memory* (RAM) model. This model is pretty close to how most off-the-shelf computers work today.

Under the RAM model of computation, we assume the following:

- **Atomic operations**: Simple mathematical ($+$, $*$, $-$, $=$) and simple functional (`if`, `call`, `store`, etc.) operations take 1 unit of time, or $O(1)$ time. We call these "atomic" operations.
- **Higher-order operations**: Loops (`for`, `while`, etc.) and subroutines are not atomic. Instead, each of these is composed of many atomic operations. Arranging a sequence of atomic operations to perform a task is the essence of programming. For example, sorting a list of numbers may be a single function call, but it is not an atomic operation: sorting a list of $n = 10$ numbers should take less time than sorting a list of $n = 10,000,000$ numbers. Similarly, how much time it takes to complete a loop, depends on how many iterations it runs.
- **Atomic data types**: Simple data types (a single boolean, or integer, or character, or real-valued number) take 1 unit of space, or $O(1)$ of space. We call these "atomic" data types. Accessing the value of an atomic data type takes 1 unit of time, and our abstract machine is equipped with an infinite amount of memory.
- **Higher-order data types**: Arrays, lists, trees, etc. are not atomic. Instead, each is composed of many atomic data types. For instance, a list of numbers can be stored together, but a list with $n = 10$ values will take less space than a list of $n = 10,000,000$ values.

All models of computation make a distinction between atomic and non-atomic operations or data types, but they may differ on *what* they consider atomic. For instance, operations like division, modular arithmetic, or generating a random variable may or may not be considered atomic. In our RAM model, you may assume these operations are also atomic.

## 2.2 Two goals of algorithm design and analysis

Algorithm design and analysis is focused primarily on two goals:

1. Algorithms that are *correct*.

   An algorithm is simply a function $f$ that takes some input $X$ and maps it to an output $Y$, i.e., $f : X \to Y$. To be useful, the $Y$ should have certain properties, and a function $f$ is called *correct* if and only if for every possible input $X$, it outputs a $Y$ with the correct properties.

   For instance, suppose $f$ is a function that sorts a list of $n$ numbers, and that $f$ is correct. The input $X$ must then always be one of the $n!$ permutations of $n$ numbers (no mixed data types allowed here), and the output $Y$ will always be the permutation of $X$ such that $y_1 \leq y_2 \leq y_3 \leq \cdots \leq y_n$ (if $f$ sorts in ascending order). If there exists some input $X'$ such that $Y' = f(X')$ is not sorted, then $f$ is not a correct algorithm.

   In this class, we will learn how to design correct algorithms, how to demonstrate that an algorithm is correct, and how to identify when an algorithm is not correct.

2. Algorithms that are *efficient*.

   A correct algorithm can still be inefficient. To be efficient, an algorithm must use *few* resources. In this class, we are concerned with two resources: *time* and *space*. How much of a resource an algorithm uses is determined by adding up all the atomic operations and expressing this quantity as a function of the size of the input $n = |X|$.

   For instance, take our correct sorting algorithm $f$. If during the middle of its operation, it pauses for $n$ seconds before continuing it sort, then clearly an algorithm $f'$ that did not pause would be more efficient in its usage of time than $f$. Similarly, if $f$ were to copy the input $X$ into a second auxiliary data structure and then delete it before continuing, an algorithm $f'$ that did not do this copy would be more efficient in its usage of space than $f$.

   In this class, we will learn how to design efficient algorithms, how to quantify how efficient an algorithm is, and how to identify when an algorithm is inefficient.

## 2.3 Adversarial algorithm analysis

To provide a rigorous *guarantee* of an algorithm's correctness or its efficiency, we use what's called "worst-case analysis," in which we analyze the algorithm's behavior under the worst possible input for correctness or for resource usage (time or space).

Identifying the algorithm input that produces the worst possible performance is a key part of this class. By the end of the class, if I describe an algorithm to you, you should be comfortable identifying these cases and using them to argument mathematically about the correctness and efficiency of the algorithm. This is the pessimistic or *adversarial* approach to algorithm analysis. Surprisingly, in many cases, we'll still be able to get fairly good bounds, and because it's a worst-case analysis, reality can not be any worse.

In some cases, we may instead consider the *average case*, which is sometimes significantly better than the worst case, but not always. (Note: the term "average" case is meaningless without some reference to a probability distribution of inputs.) *Best case* analysis is inherently optimistic, and thus is not typically useful for providing algorithmic guarantees.

## 2.4 Asymptotic analysis

Unless otherwise specified, we will care only about the asymptotic behavior of how much of a resource (time or space) an algorithm uses.

This approach simplifies our work and concisely communicates the core differences between different algorithms. This class assumes that you already are familiar with the basics of asymptotic

analysis and are familiar with the meaning of $O(.)$, $\Theta(.)$, $\Omega(.)$.

The asymptotic behavior of an algorithm should be expressed as a function of the input size, which is conventionally denoted by $n$. (In some cases, there are multiple input parameters, e.g., in a graph $G = \{V, E\}$ where $|V| = n$ and $|E| = m$; in this case, the asymptotic behavior can be a function of multiple parameters.)

As a rule of thumb, in asymptotic analysis, we are not interested in constants, either multiplicative or additive, so $5n + 2$, $n$ and $10000n$ are all $O(n)$ and $10^4$, $2^{50}$ and 4 are all $O(1)$.

For big-$O$, we are interested mainly in the leading term (fastest growing) of the function, and thus we can neglect slower growing functions or trailing terms (but not multiplicative functions). Thus, $n^2 + n + \log n$, $10n^2 + n - \sqrt{n}$ and $n^2 + 10^6 n$ are all $O(n^2)$.

Section 2.2 in the textbook covers asymptotic analysis. Read it. Know it.

Here are some analogies and examples you can use in thinking about asymptotic notation.

| | | |
|---|---|---|
| $O$ | "$\leq$" | This algorithm is $O(n^2)$, that is, worst case is $\Theta(n^2)$. |
| $\Theta$ | "$=$" | This algorithm is $\Theta(n)$, that is, best and worst case are $\Theta(n)$. |
| $\Omega$ | "$\geq$" | Any comparison-based algorithm for sorting is $\Omega(n \log n)$. |
| $o$ | "$<$" | Can you write an algorithm for sorting that is $o(n^2)$? |
| $\omega$ | "$>$ " | This algorithm is not linear, it can take time $\omega(n)$. |

Here are some more mathematical facts you should memorize:

1. L'Hopital's rule:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\frac{\partial}{\partial n} f(n)}{\frac{\partial}{\partial n} g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

2. $(x^y)^z = x^{yz}$
3. $x^y x^z = x^{y+z}$
4. $\log_x y = z \implies x^z = y$
5. $x^{\log_x y} = y$ by definition
6. $\log(xy) = \log x + \log y$
7. $\log(x^c) = c \log x$
8. $\log_c(x) = \log x / \log c$

## 3 On your own

1. Read Chapters 1 and 2 in Kleinberg–Tardos.

# 4 Addendum: Proofs by induction

*Induction* is a simple and powerful technique for proving mathematical claims, and is one of the most common tools in algorithm analysis, precisely because of these properties. When we invoke this approach to proving a claim, we must state exactly what variable or property we are using induction *on*.

Consider the example above, the sum of the first $n$ positive integers, $S(n) = 1 + 2 + 3 + \cdots + n$. We claim that a closed form for this sum is

$$S(n) = \frac{n(n+1)}{2} \ . \tag{1}$$

We will mathematically prove, via induction on $n$, that $n(n+1)/2$ is the correct closed-form solution.[1] A proof by induction has two parts:

- **Base case**: in which we verify that $S(n)$ is correct for the smallest possible value, in this case $n = 1$. (If the claim is about some recurrence relation, the base cases will often be given to you, e.g., $T(0) = c_1$ and $T(1) = c_2$, for two base cases where $c_1$ and $c_2$ are constants.) That is,

$$S(1) = \frac{1(1+1)}{2} = 1 \ .$$

- **Inductive step**: in which we (i) assume that $S(n)$ holds for an arbitrary input of size $n$ and then (ii) prove that it also holds for $n + 1$. In practice, this amounts to taking Eq. 1, or its equivalent in your problem of choice, and replacing each instance of $n$ with an instance of $n + 1$ and the simplifying:

$$\begin{aligned} S(n+1) &= \frac{(n+1)((n+1)+1)}{2} \\ &= \frac{(n+1)(n+2)}{2} \\ &= \frac{n^2+3n+2}{2} \\ &= \frac{n^2+n+2n+2}{2} \\ &= \frac{n(n+1)}{2} + \frac{2n+2}{2} \\ &= S(n) + (n+1) \ , \end{aligned}$$

[1] Here, there is only one possible choice, but for more complex algorithms, there will be several choices. In those cases, our job is to choose correctly, so as to make the induction straightforward. A hint that you have chosen incorrectly is that the induction is hard to push through.

where in the final step we have used the definition of $S(n)$ to reduce a more complicated expression into a simple one completes the inductive step. (In some problems, there will be cases to consider, e.g., $n \geq c$ for $c$ constant and $n < c$, and in that situation, the inductive step must be applied to each.)

In this class, we aim to present more formal proofs. Thus, rather than the more wordy, thinking-out-loud version above, here is a compact, formal proof, which states the claim precisely, states the assumptions clearly, and shows exactly what is necessary to understand the proof.

*Theorem: The sum of the first $n$ positive integers is $S(n) = n(n+1)/2$.*

*Proof*: Let $n$ be an arbitrary positive integer, and assume inductively that $S(n) = n(n+1)/2$. The base case $n = 1$ is $S(1) = 1(1+1)/2 = 1$, and for any $n \geq 1$ we have

$$\begin{aligned} S(n+1) &= \frac{(n+1)((n+1)+1)}{2} && \big[\text{definition of } S(n+1)\big] \\ &= \frac{n(n+1)}{2} + \frac{2n+2}{2} && \big[\text{algebra}\big] \\ &= S(n) + (n+1) && \big[\text{inductive hypothesis with } S(n)\big] \ , \end{aligned}$$

□

If you would like to see more examples of proofs by induction or get a deeper mathematical explanation of why and how they work, see the excellent notes by Jeff Erickson at UIUC:
`http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/98-induction.pdf`

## 5 Addendum: Thinking algorithmically to find a duplicate

As a small aside, consider the problem of finding a duplicate value in an array of otherwise unique values:

*You're given a list of $n+1$ real-valued numbers $\{x_i\}$, stored in an array $A$. Exactly $n$ of the values of unique, but one is a duplicate. Describe an efficient algorithm to find it and give its asymptotic running time.*

To again illustrate how to think carefully about a project like this, we'll consider two solutions, one naïve and one efficient.

## A naïve approach

Let's assume that the values in the list $\{x_i\}$ are integers on the unbounded interval $x_i \in (-\infty, \infty)$, and let's apply the *histogram* method, in which we allocate a second array $B$ to store the histogram, or the frequency $f_i$ of the $i$th integer.

In order to allocate $B$, we must first identify the range that the $x_i$ values span since $B$ will contain $m = \max_i A[i] - \min_i A[i] + 1$ elements. We could do this by running over each element in $A$ and keeping track of the largest and smallest values we observe, denoted $maxA$ and $minA$ respectively. This step of our algorithm takes $\Theta(n)$ time. Once we've allocated $B$, we then loop over the elements of $A$ a second time and increment the corresponding count variable in $B$, i.e, `B[minA+A[i]]++` (why do we index into $B$ in this way?). When this is finished, the elements of $B$ are a histogram of the values in $A$. Finally, we loop over $B$ looking for any element with `B[i]>1`; alternatively, if we are confident there is only a single duplicate, before we increment a count variable, we could first check if it has previously been incremented; if so, we have found the duplicate and we can terminate. (Does this second strategy change the worst-case running time? Where would the adversary place the duplicate in $A$ in order to guarantee the worst case?)

How long does this algorithm take? How much memory does it take? Remember that we must assume the adversarial model of algorithm analysis—what's the worst kind of input $A$ that the adversary could give us? The answer is that this algorithm takes time $O(maxA - minA)$ and that there is no bound on how large $maxA - minA$ could be. Thus, the adversary could choose the an input in which $n$ of the values of consecutive integers starting at 1, but with $maxA = n^n$ or worse. Because the running time depends mainly on the time to allocate $B$, there is no limit to how slow this algorithm could be.

## An efficient approach

A better solution is to dispense with the histogram solution completely and instead use a sorting approach. As we discussed in class, we could use an efficient sorting algorithm like QuickSort or MergeSort to reorder the $\{x_i\}$ in ascending order. This step takes $O(n \log n)$ time. To find the duplicate, we simply examine each element in turn and compare it with the preceding element; if this comparison is ever true, we return either of the two elements. That is, for each $i$, we ask `A[i]==A[i+1]` and we return `A[i]` if the comparison is true.

This algorithm can be naturally generalized to solve the problem of finding the *mode* of the list of numbers, i.e., the most frequent value, not just a single duplicate. Again, sort the input array. Now, store the first element `s=A[1]`, initialize a counter $c_s = 1$ (which will count the frequency of the value stored in $s$), initialize a value `t=NULL` and a second counter $c_t = -\infty$. The idea is to iterate through the list using $c_s$ to measure the frequency of the value stored in $s$; because the

array is sorted, all of these values much occur together. When we encounter the first value that is different from $s$, we check to see whether the $c_t < c_s$, i.e., if the value stored in $s$ occurs more frequently than the value stored in $t$ (our old "most common" value). Is it does, then we store the new candidate for most common value ($t \leftarrow s$ and $c_t \leftarrow c_s$), increment $i$ and set $s = A[i]$ and $c_s = 1$ and proceed.

At the end of the loop, we check if $c_t < c_s$; if so, we return $s$ as the mode; if not, we return $t$. (Can you prove that this algorithm is correct? That is, that it returns the correct answer on all inputs? What is the worst case input?)

## A general solution

The sorting approach solves the problem of finding the modal value, but it does not solve the more general problem of finding the frequencies of all the values. Fortunately, from a computational point of view, finding the frequencies of *all* the values is *not* more expensive than finding the frequency of only one value. Formally, I claim that the time required to find the most frequent value and the time required to find the frequencies of all values (the histogram) are asymptotically equal, and both take $O(n \log n)$ time. How can this be?

It should be clear that if I can construct the histogram in $O(n \log n)$ time, I can return the $k$th most frequent value by simply sorting the values by their frequencies and then returning the $k$th value. Because sorting takes $O(n \log n)$ time, it is no more expensive than the construction itself, which makes this step "free" with respect to the total asymptotic running time. (In fact, I can do a constant number of any operations with that running time for "free" and not change the asymptotics.) Thus, it only remains to show that I can construct the histogram in $O(n \log n)$ time.

The naïve histogram method suffers from allocating an array that spans the interval $[minA, maxA]$, which could be expensive because $maxA$ could be extremely large. In our general solution, we will dispense with the second array $B$ and instead count only the values that actually occur. In this way, we can avoid the problem identified above, which is that the values of $\{x_i\}$ may be sparsely distributed over the range $[minA, maxA]$, and the running time above depends on just how sparse they are. If we are sloppy, a clever adversary could make our algorithm take an infinite amount of time by simply adding a single value $y$ to the list where $y$ is infinitely large. To do this, we will use a self-balancing binary search tree data structure to implement a *sparse vector*.

Recall that a self-balancing binary search tree, like an AVL tree or a red-black tree (Chapter 13), stores a set of tuples of the form `(key,value)`. For our purposes, let the "keys" be the observed values $x_i$ and the "values" be their frequencies in $A$. To begin, we initialize our sparse vector to be empty and then begin iterating over the elements of $A$, in order. For each element $x_i$, we perform a `find(x_i)` operation on our sparse vector; if it returns a `NULL`, then we know that we

have not encountered this value before and then call `insert(x_i,1)` to add it to the histogram; otherwise, it returns a value $f$, which is the frequency of $x_i$ and we then call `delete(x_i)` followed by `insert(x_i,f+1)` in order to increment its frequency.

Find, insert and delete operations for self-balancing binary search trees all take $O(\log n)$ time, and thus for $n$ values, this algorithm takes $O(n \log n)$ time to complete and $O(n)$ additional memory. To read out the histogram in sorted order, we then do an in-order traversal (a.k.a., depth-first search) through the tree structure and store the results in a new auxiliary array $B$. Thus, constructing the histogram can be done in $O(n \log n)$ time.